Digital Logic Design

4190.201.001

2010 Spring Semester

6. Case Studies in Combinational Logic Design

Naehyuck Chang
Dept. of EECS/CSE
Seoul National University
naehyuck@snu.ac.kr

Combinational logic design case studies

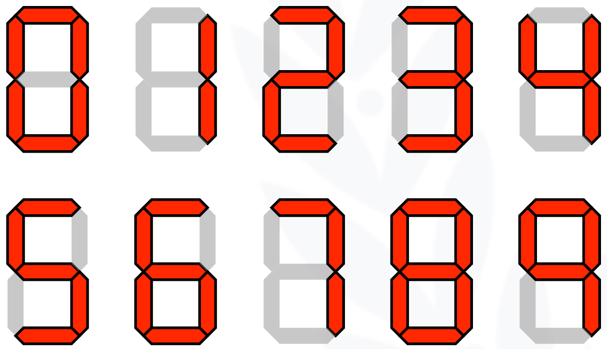
- General design procedure
- Case studies
  - BCD to 7-segment display controller
  - Logical function unit
  - Process line controller
  - Calendar subsystem
- Arithmetic circuits
  - Integer representations
  - Addition/subtraction
  - Arithmetic/logic units

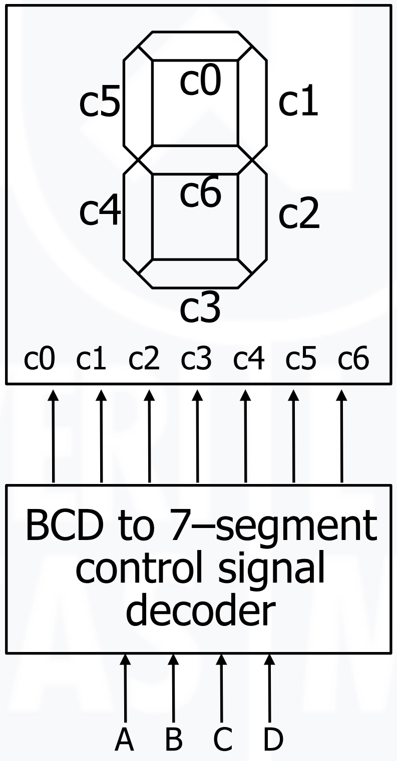
General design procedure for combinational logic

- Step 1. Understand the problem
  - What is the circuit supposed to do?
  - Write down inputs (data, control) and outputs
  - Draw block diagram or other picture
- Step 2. Formulate the problem using a suitable design representation
  - Truth table or waveform diagram are typical
  - May require encoding of symbolic inputs and outputs
- Step 3. Choose implementation target
  - ROM, PAL, PLA
  - Mux, decoder and OR-gate
  - Discrete gates
- Step 4. Follow implementation procedure
  - K-maps for two-level, multi-level
  - Design tools and hardware description language (e.g., Verilog)

BCD to 7-segment display controller

- Understanding the problem
  - input is a 4 bit bcd digit (A, B, C, D)
  - output is the control signals
    for the display (7 outputs C0 – C6)
- Block diagram





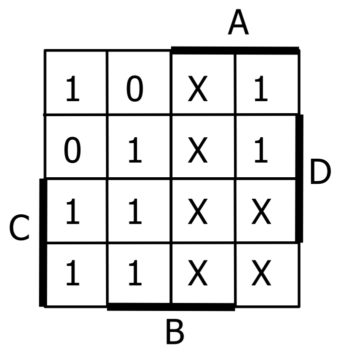
Formalize the problem

- Truth table
  - Show don't cares
- Choose implementation target
  - If ROM, we are done
  - Don't cares imply PAL/PLA may be attractive
- Follow implementation procedure
  - Minimization using K-maps

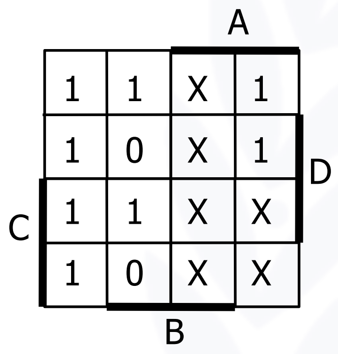
A	B	C	D	C0	C1	C2	C3	C4	C5	C6
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	0	0	1	1
1	0	1	–	–	–	–	–	–	–	–
1	1	–	–	–	–	–	–	–	–	–

Implementation as minimized sum-of-products

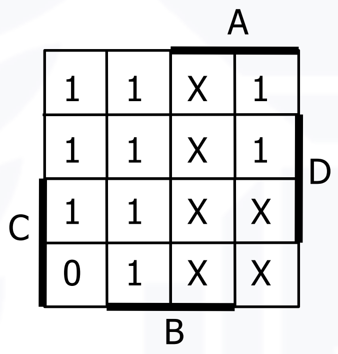
- 15 unique product terms when minimized individually



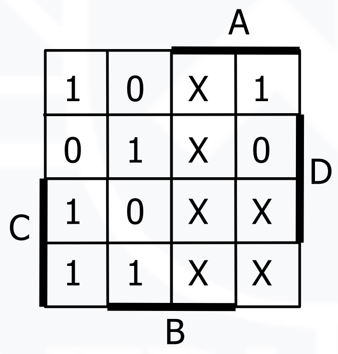
1	0	X	1
0	1	X	1
1	1	X	X
1	1	X	X



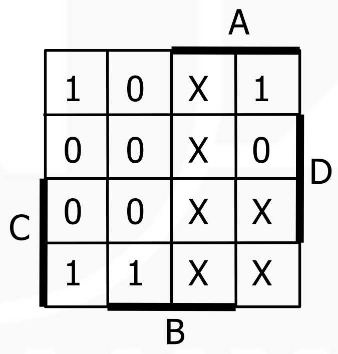
1	1	X	1
1	0	X	1
1	1	X	X
1	0	X	X



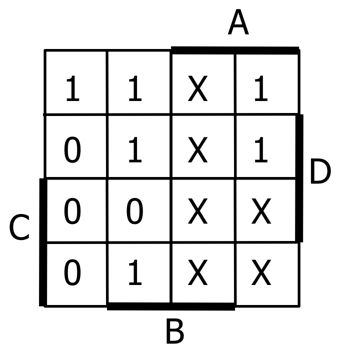
1	1	X	1
1	1	X	1
1	1	X	X
0	1	X	X



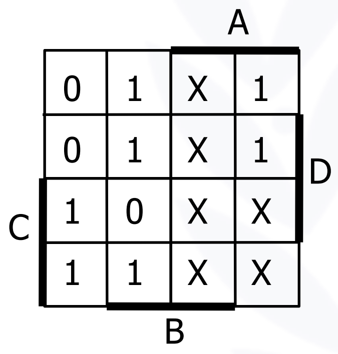
1	0	X	1
0	1	X	0
1	0	X	X
1	1	X	X



1	0	X	1
0	0	X	0
0	0	X	X
1	1	X	X



1	1	X	1
0	1	X	1
0	0	X	X
0	1	X	X



0	1	X	1
0	1	X	1
1	0	X	X
1	1	X	X

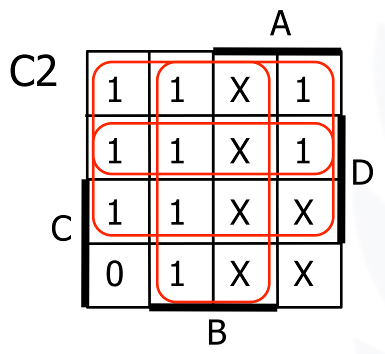
C0 = A + B D + C + B' D'
C1 = C' D' + C D + B'
C2 = B + C' + D
C3 = B' D' + C D' + B C' D + B' C
C4 = B' D' + C D'
C5 = A + C' D' + B D' + B C'
C6 = A + C D' + B C' + B' C

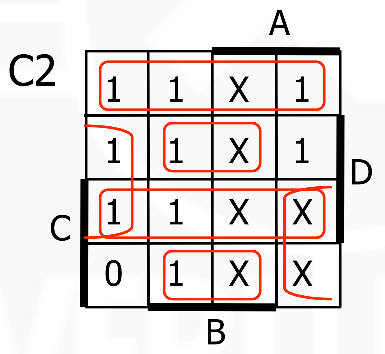
Embedded Low-Power Laboratory

Monday, March 15, 2010

Implementation as minimized S-o-P (cont'd)

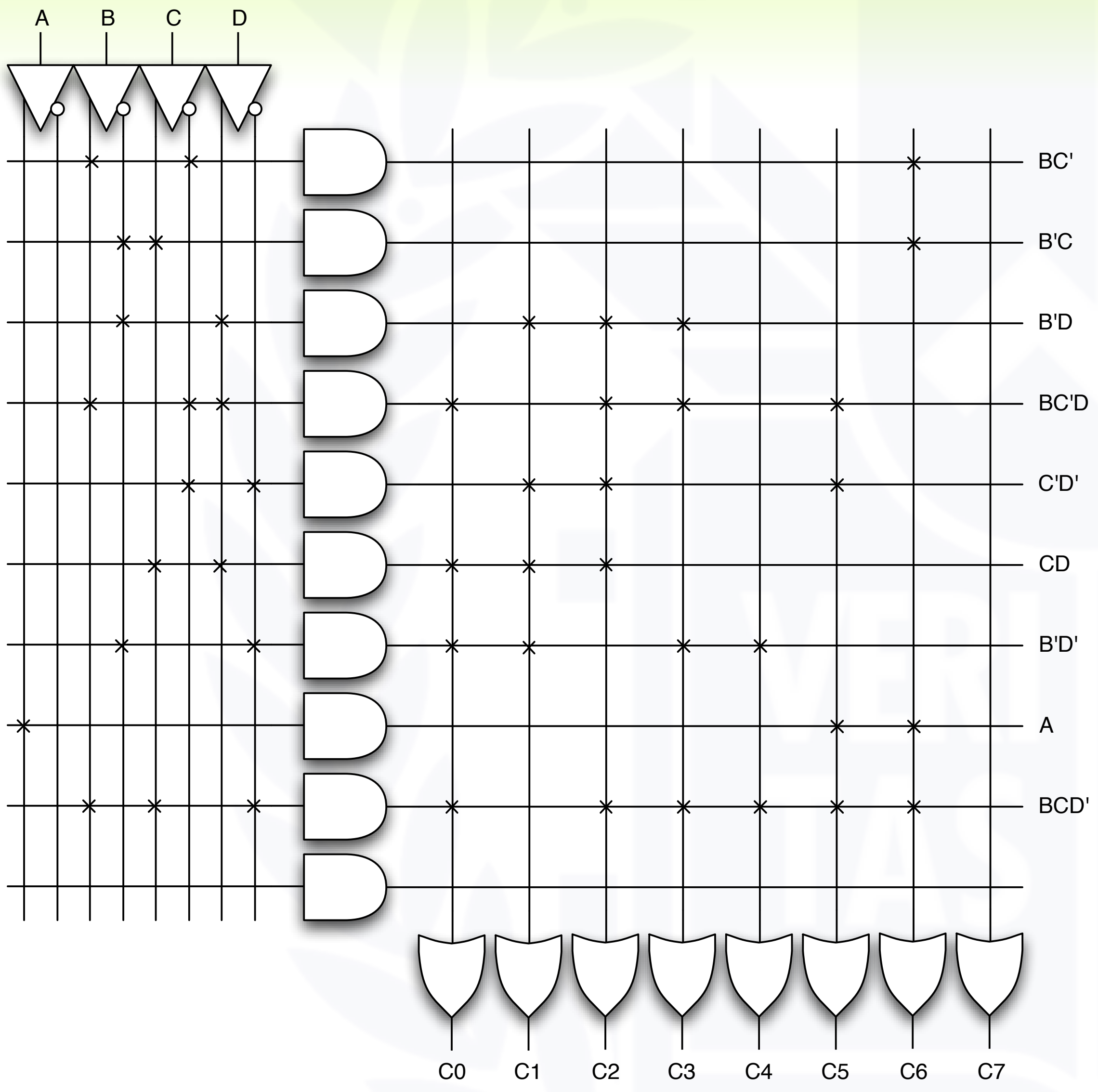
- Can do better
  - 9 unique product terms (instead of 15)
  - Share terms among outputs
  - Each output not necessarily in minimized form





C0 = A + B D + C + B' D'	C0 = B C' D + C D + B' D' + B C D' + A
C1 = C' D' + C D + B'	C1 = B' D + C' D' + C D + B' D'
C2 = B + C' + D	C2 = B' D + B C' D + C' D' + C D + B C D'
C3 = B' D' + C D' + B C' D + B' C	C3 = B C' D + B' D + B' D' + B C D'
C4 = B' D' + C D'	C4 = B' D' + B C D'
C5 = A + C' D' + B D' + B C'	C5 = B C' D + C' D' + A + B C D'
C6 = A + C D' + B C' + B' C	C6 = B' C + B C' + B C D' + A

PLA implementation



Embedded Low-Power Laboratory

PAL implementation vs. Discrete gate implementation

- Limit of 4 product terms per output
  - Decomposition of functions with larger number of terms
  - Do not share terms in PAL anyway
    (although there are some with some shared terms)

C2 = B + C' + D
C2 = B' D + B C' D + C' D' + C D + B C D'
C2 = B' D + B C' D + C' D' + W ← need another input and another output
W = C D + B C D'

- Decompose into multi-level logic (hopefully with CAD support)
  - Find common sub-expressions among functions

C0 = C3 + A' B X' + A D Y
C1 = Y + A' C5' + C' D' C6
C2 = C5 + A' B' D + A' C D
C3 = C4 + B D C5 + A' B' X'
C4 = D' Y + A' C D'
C5 = C' C4 + A Y + A' B X
C6 = A C4 + C C5 + C4' C5 + A' B' C

X = C' + D'
Y = B' C'

Embedded Low-Power Laboratory

Logical function unit

- Multi-purpose function block
  - 3 control inputs to specify operation to perform on operands
  - 2 data inputs for operands
  - 1 output of the same bit-width as operands

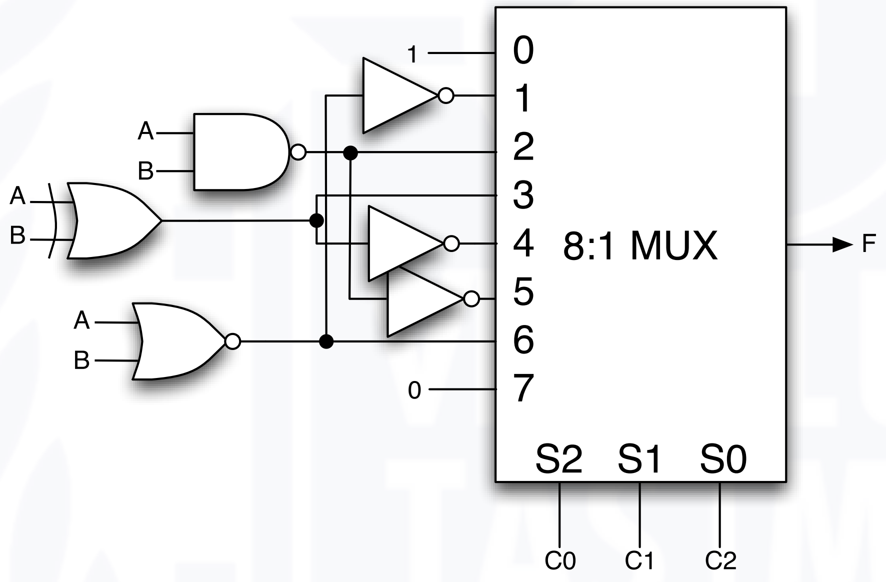
C0	C1	C2	Function	Comments
0	0	0	1	always 1
0	0	1	A + B	logical OR
0	1	0	(A • B)'	logical NAND
0	1	1	A xor B	logical xor
1	0	0	A xnor B	logical xnor
1	0	1	A • B	logical AND
1	1	0	(A + B)'	logical NOR
1	1	1	0	always 0

3 control inputs: C0, C1, C2
2 data inputs: A, B
1 output: F

Formalize the problem

C0	C1	C2	A	B	F
0	0	0	0	0	1
0	0	0	0	1	1
0	0	0	1	0	1
0	0	0	1	1	1
0	0	1	0	0	0
0	0	1	0	1	1
0	0	1	1	0	1
0	0	1	1	1	1
0	1	0	0	0	1
0	1	0	0	1	1
0	1	0	1	0	1
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	0	1	1
0	1	1	1	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	0	0	1	0
1	0	0	1	0	0
1	0	0	1	1	1
1	0	1	0	0	0
1	0	1	0	1	0
1	0	1	1	0	0
1	0	1	1	1	1
1	1	0	0	0	1
1	1	0	0	1	0
1	1	0	1	0	0
1	1	0	1	1	0
1	1	1	0	0	0
1	1	1	0	1	0
1	1	1	1	0	0
1	1	1	1	1	0

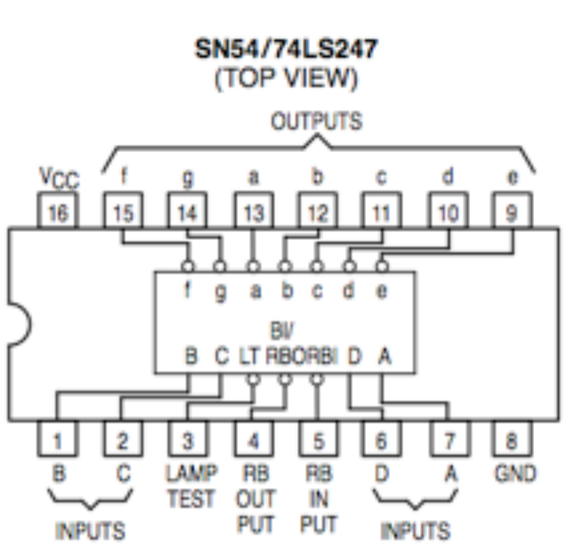
Choose implementation technology
5-variable K-map to discrete gates
multiplexor implementation

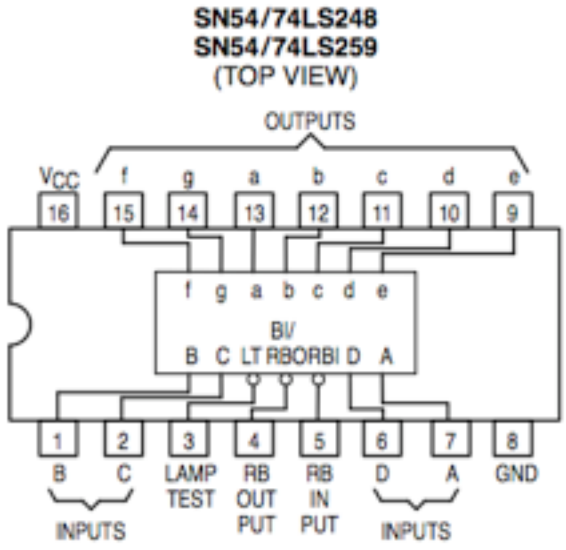


Monday, March 15, 2010

74LS247

SN54/74LS247 • SN54/74LS248 • SN54/74LS249





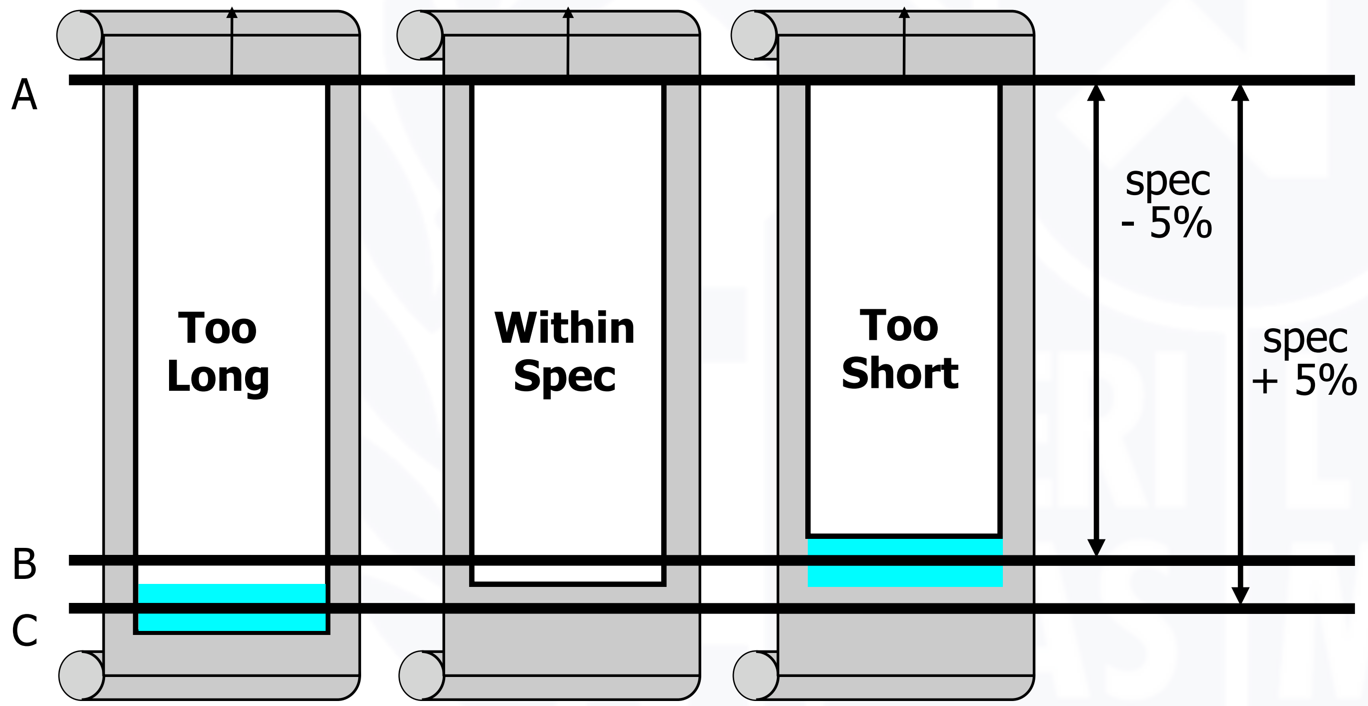
Embedded Low-Power Laboratory

Production line control

- Rods of varying length (+/-10%) travel on conveyor belt
  - Mechanical arm pushes rods within spec (+/-5%) to one side
  - Second arm pushes rods too long to other side
  - Rods that are too short stay on belt
  - 3 light barriers (light source + photocell) as sensors
  - Design combinational logic to activate the arms
- Understanding the problem
  - Inputs are three sensors
  - Outputs are two arm control signals
  - Assume sensor reads "1" when tripped, "0" otherwise
  - Call sensors A, B, C

Sketch of problem

- Position of sensors
  - A to B distance = specification – 5%
  - A to C distance = specification + 5%



Embedded Low-Power Laboratory

Formalize the problem

- Truth table
  - Show don't cares

A	B	C	Function
0	0	0	do nothing
0	0	1	do nothing
0	1	0	do nothing
0	1	1	do nothing
1	0	0	too short
1	0	1	don't care
1	1	0	in spec
1	1	1	too long

Logic implementation now straightforward
just use three 3-input AND gates

"too short" = AB'C'
(only first sensor tripped)

"in spec" = A B C'
(first two sensors tripped)

"too long" = A B C
(all three sensors tripped)

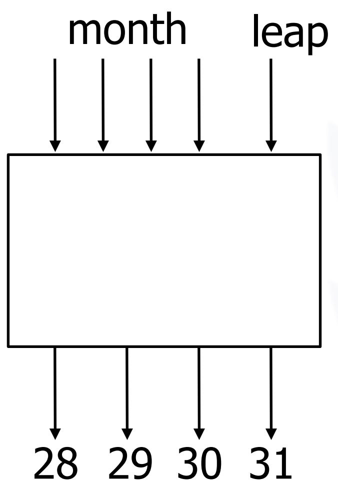
Calendar subsystem

- Determine number of days in a month (to control watch display)
  - Used in controlling the display of a wrist-watch LCD screen
  - Inputs: month, leap year flag
  - Outputs: number of days
- Use software implementation to help understand the problem

```
integer number_of_days ( month, leap_year_flag) {
     switch (month) {
          case  1: return (31);
          case  2: if (leap_year_flag == 1)
                      then return (29)
                      else return (28);
          case  3: return (31);
          case  4: return (30);
          case  5: return (31);
          case  6: return (30);
          case  7: return (31);
          case  8: return (31);
          case  9: return (30);
          case 10: return (31);
          case 11: return (30);
          case 12: return (31);
          default: return (0);
     }
}
```

Formalize the problem

- Encoding:
  - Binary number for month: 4 bits
  - 4 wires for 28, 29, 30, and 31
    one-hot – only one true at any time
- Block diagram:



month	leap	28	29	30	31
0000	–	–	–	–	–
0001	–	0	0	0	1
0010	0	1	0	0	0
0010	1	0	1	0	0
0011	–	0	0	0	1
0100	–	0	0	1	0
0101	–	0	0	0	1
0110	–	0	0	1	0
0111	–	0	0	0	1
1000	–	0	0	0	1
1001	–	0	0	1	0
1010	–	0	0	0	1
1011	–	0	0	1	0
1100	–	0	0	0	1
1101	–	–	–	–	–
111–	–	–	–	–	–

Embedded Low-Power Laboratory

Choose implementation target and perform mapping

- Discrete gates
  - 28 = m8′ m4′ m2 m1′ leap′
  - 29 = m8′ m4′ m2 m1′ leap
  - 30 = m8′ m4 m1′ + m8 m1
  - 31 = m8′ m1 + m8 m1′
- Can translate to S-o-P or P-o-S

month	leap	28	29	30	31
0000	–	–	–	–	–
0001	–	0	0	0	1
0010	0	1	0	0	0
0010	1	0	1	0	0
0011	–	0	0	0	1
0100	–	0	0	1	0
0101	–	0	0	0	1
0110	–	0	0	1	0
0111	–	0	0	0	1
1000	–	0	0	0	1
1001	–	0	0	1	0
1010	–	0	0	0	1
1011	–	0	0	1	0
1100	–	0	0	0	1
1101	–	–	–	–	–
111–	–	–	–	–	–

Leap year flag

- Determine value of leap year flag given the year
  - For years after 1582 (Gregorian calendar reformation),
  - Leap years are all the years divisible by 4,
  - Except that years divisible by 100 are not leap years,
  - But years divisible by 400 are leap years.
- Encoding the year:
  - Binary – easy for divisible by 4,
    but difficult for 100 and 400 (not powers of 2)
  - BCD – easy for 100,
    but more difficult for 4, what about 400?
- Parts:
  - Construct a circuit that determines if the year is divisible by 4
  - Construct a circuit that determines if the year is divisible by 100
  - Construct a circuit that determines if the year is divisible by 400
  - Combine the results of the previous three steps to yield the leap year flag

Activity: divisible-by-4 circuit

- BCD coded year
  - YM8 YM4 YM2 YM1 – YH8 YH4 YH2 YH1 – YT8 YT4 YT2 YT1 – YO8 YO4 YO2 YO1
- Only need to look at low-order two digits of the year
  all years ending in 00, 04, 08, 12, 16, 20, etc. are divisible by 4
  - If tens digit is even, then divisible by 4 if ones digit is 0, 4, or 8
  - If tens digit is odd, then divisible by 4 if the ones digit is 2 or 6.
- Translates into the following Boolean expression
  (where YT1 is the year's tens digit low-order bit,
  YO8 is the high-order bit of year's ones digit, etc.):
  - YT1' (YO8' YO4' YO2' YO1' + YO8' YO4 YO2' YO1' + YO8 YO4' YO2' YO1')
    + YT1 (YO8' YO4' YO2 YO1' + YO8' YO4 YO2 YO1')
- Digits with values of 10 to 15 will never occur, simplify further to yield:
  - YT1' YO2' YO1' + YT1 YO2 YO1'

Embedded Low-Power Laboratory

Divisible-by-100 and divisible-by-400 circuits

- Divisible-by-100 just requires checking that all bits of two low-order digits are all 0:
  - YT8′ YT4′ YT2′ YT1′ • YO8′ YO4′ YO2′ YO1′
- Divisible-by-400 combines the divisible-by-4 (applied to the thousands and hundreds digits) and divisible-by-100 circuits
  - (YM1′ YH2′ YH1′ + YM1 YH2 YH1′) • (YT8′ YT4′ YT2′ YT1′ • YO8′ YO4′ YO2′ YO1′)

Combining to determine leap year flag

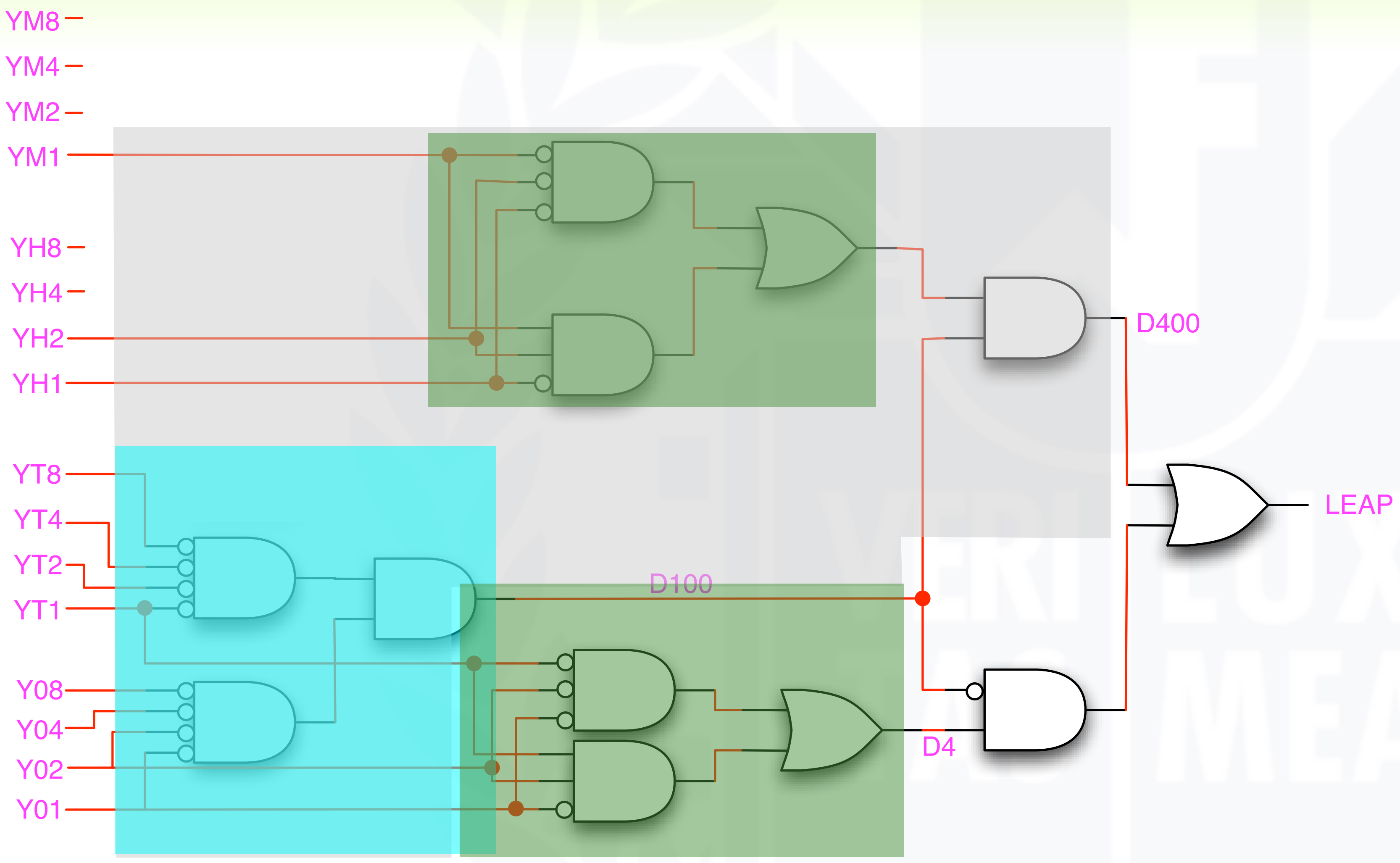
- Label results of previous three circuits: D4, D100, and D400

$$\text{leap\_year\_flag} = D4\,(D100 \bullet D400')'$$

$$= D4 \bullet D100' + D4 \bullet D400$$

$$= D4 \bullet D100' + D400$$

Implementation of leap year flag



Arithmetic circuits

- Excellent examples of combinational logic design
- Time vs. space trade-offs
  - Doing things fast may require more logic and thus more space
  - Example: carry lookahead logic
- Arithmetic and logic units
  - General-purpose building blocks
  - Critical components of processor datapaths
  - Used within most computer instructions

Number systems

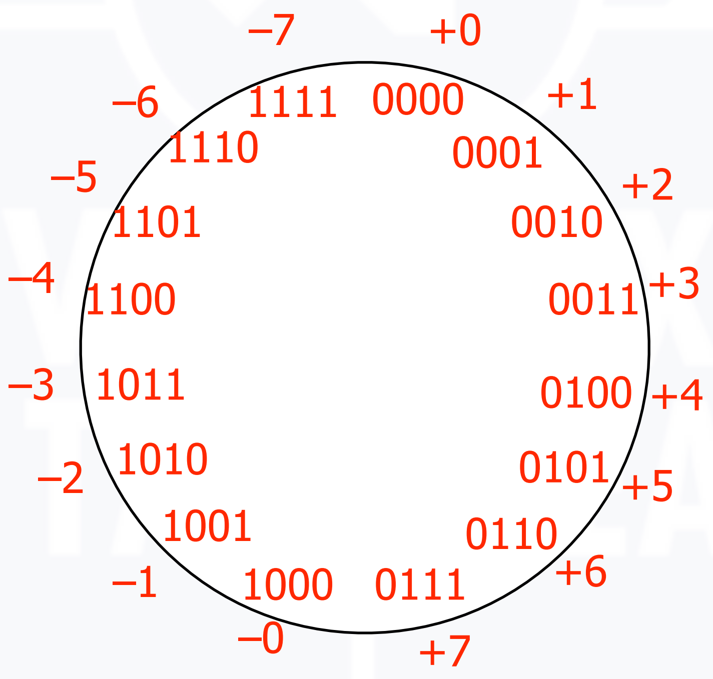
- Representation of positive numbers is the same in most systems
- Major differences are in how negative numbers are represented
- Representation of negative numbers come in three major schemes
  - Sign and magnitude
  - 1s complement
  - 2s complement
- Assumptions
  - We'll assume a 4 bit machine word
  - 16 different values can be represented
  - Roughly half are positive, half are negative

Sign and magnitude

- One bit dedicate to sign (positive or negative)
  - Sign: 0 = positive (or zero), 1 = negative
- Rest represent the absolute value or magnitude
  - Three low order bits: 0 (000) thru 7 (111)
- Range for n bits
  - +/– 2n–1 –1 (two representations for 0)
- Cumbersome addition/subtraction
  - Must compare magnitudes to determine sign of result

0 100 = + 4

1 100 = – 4



Embedded Low-Power Laboratory

Monday, March 15, 2010

1s complement

- If N is a positive number, then the negative of N (its 1s complement or N') is N' = $(2^n - 1) - N$
  - Example: 1s complement of 7

$$
\begin{array}{lcr}
2^4 & = & 10000 \\
1 & = & 00001 \\
\hline
2^4 - 1 & = & 1111 \\
7 & = & 0111 \\
\hline
 & & 1000
\end{array}
$$

1000 = −7 in 1s complement form

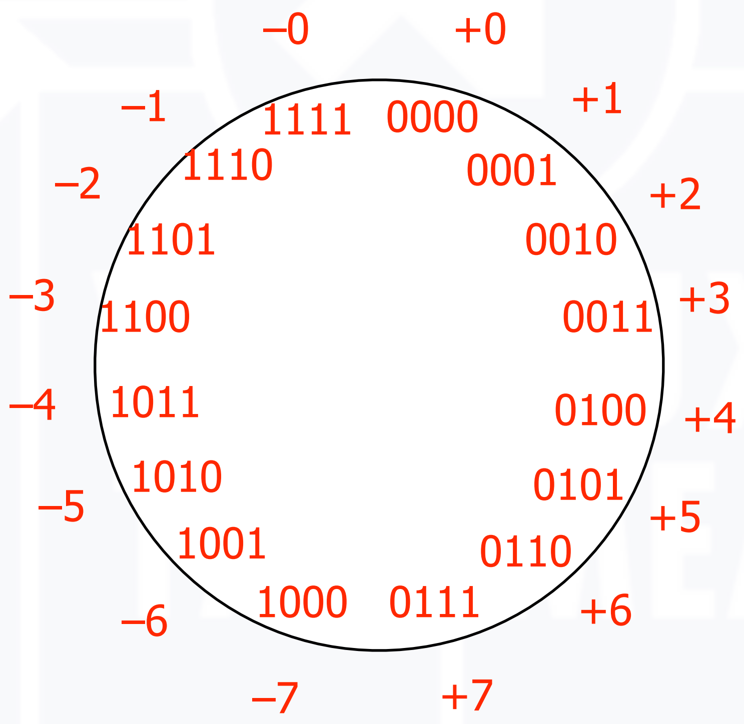
  - Shortcut: simply compute bit-wise complement (0111 -> 1000)

1s complement (cont'd)

- Subtraction implemented by 1s complement and then addition
- Two representations of 0
    - Causes some complexities in addition
- High-order bit can act as sign bit

0 100 = + 4

1 011 = – 4

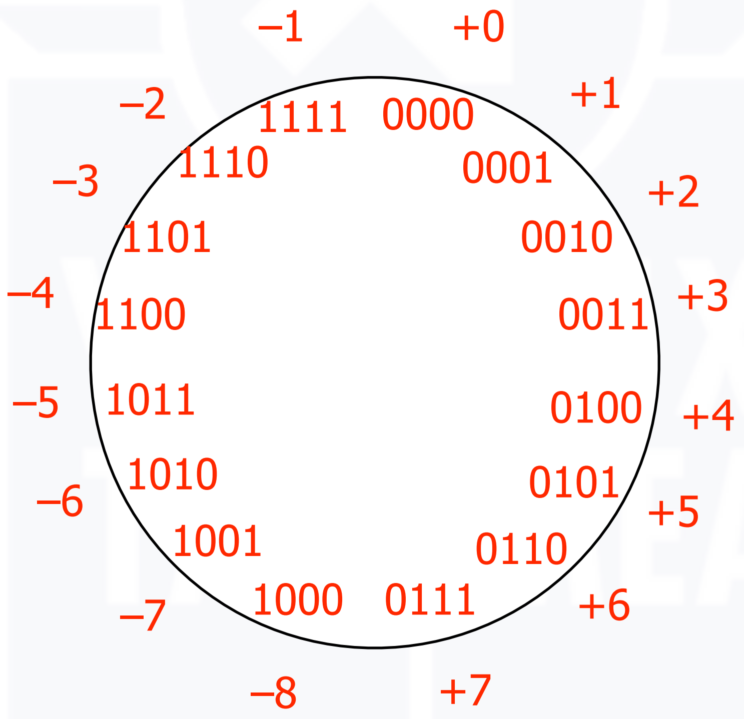


2s complement

- 1s complement with negative numbers shifted one position clockwise
  - Only one representation for 0
  - One more negative number than positive numbers
  - High-order bit can act as sign bit

0 100 = + 4

1 100 = – 4



Embedded Low-Power Laboratory

2s complement (cont'd)

- If N is a positive number, then the negative of N (its 2s complement or N*) is N* = 2n – N
  - Example: 2s complement of 7

$$
\begin{array}{lrcr}
 & 2^4 & = & 10000 \\
\text{subtract} & 7 & = & \underline{0111} \\
 & & & 1001 \quad = \text{repr. of } -7
\end{array}
$$

  - Example: 2s complement of –7

$$
\begin{array}{lrcr}
 & 2^4 & = & 10000 \\
\text{subtract} & -7 & = & \underline{1001} \\
 & & & 0111 \quad = \text{repr. of } 7
\end{array}
$$

  - Shortcut: 2s complement = bit-wise complement + 1
    - 0111 -> 1000 + 1 -> 1001 (representation of -7)
    - 1001 -> 0110 + 1 -> 0111 (representation of 7)

Embedded Low-Power Laboratory

2s complement addition and subtraction

- Simple addition and subtraction
  - Simple scheme makes 2s complement the virtually unanimous choice for integer number systems in computers

$$\begin{array}{rr}
4 & 0100 \\
+\,3 & 0011 \\
\hline
7 & 0111
\end{array}
\qquad\qquad
\begin{array}{rr}
-\,4 & 1100 \\
+\,(-\,3) & 1101 \\
\hline
-\,7 & 11001
\end{array}$$

$$\begin{array}{rr}
4 & 0100 \\
-\,3 & 1101 \\
\hline
1 & 10001
\end{array}
\qquad\qquad
\begin{array}{rr}
-\,4 & 1100 \\
+\,3 & 0011 \\
\hline
-\,1 & 1111
\end{array}$$

Embedded Low-Power Laboratory

Why can the carry-out be ignored?

- Can't ignore it completely
  - Needed to check for overflow (see next two slides)
- When there is no overflow, carry-out may be true but can be ignored

– M + N when N > M:

M* + N = (2n – M) + N = 2n + (N – M)

ignoring carry-out is just like subtracting 2n

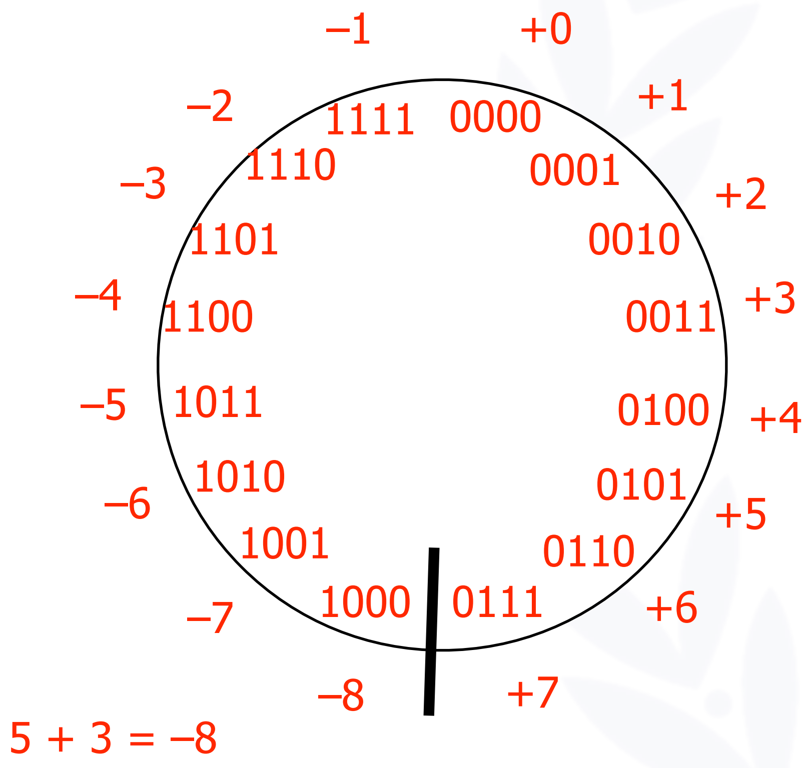
– M + – N where N + M ≤ 2n–1

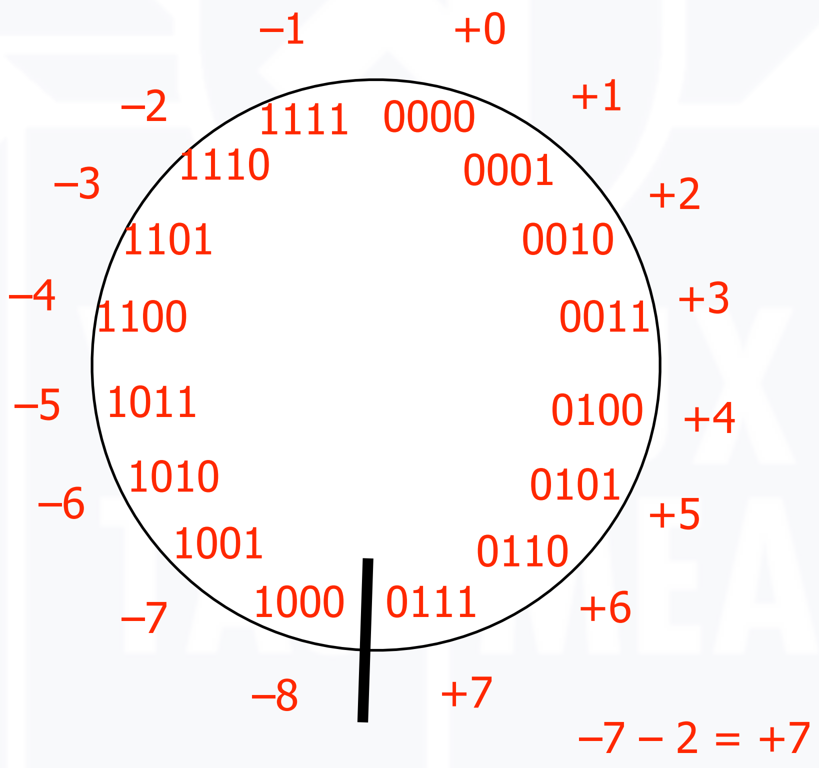
(– M) + (– N) = M* + N* = (2n– M) + (2n– N) = 2n – (M + N) + 2n

ignoring the carry, it is just the 2s complement representation for – (M + N)

Overflow in 2s complement addition/subtraction

- Overflow conditions
  - Add two positive numbers to get a negative number
  - Add two negative numbers to get a positive number





Embedded Low-Power Laboratory

Overflow conditions

- Overflow when carry into sign bit position is not equal to carry-out

$$\begin{array}{r}
5 \\
3 \\
\hline
-8
\end{array}
\qquad
\begin{array}{r}
0\ 1\ 1\ 1\ \ \\
0\ 1\ 0\ 1 \\
0\ 0\ 1\ 1 \\
\hline
1\ 0\ 0\ 0
\end{array}$$

overflow

$$\begin{array}{r}
-7 \\
-2 \\
\hline
7
\end{array}
\qquad
\begin{array}{r}
1\ 0\ 0\ 0\ \ \\
1\ 0\ 0\ 1 \\
1\ 1\ 1\ 0 \\
\hline
1\ 0\ 1\ 1\ 1
\end{array}$$

overflow

$$\begin{array}{r}
5 \\
2 \\
\hline
7
\end{array}
\qquad
\begin{array}{r}
0\ 0\ 0\ 0\ \ \\
0\ 1\ 0\ 1 \\
0\ 0\ 1\ 0 \\
\hline
0\ 1\ 1\ 1
\end{array}$$

no overflow

$$\begin{array}{r}
-3 \\
-5 \\
\hline
-8
\end{array}
\qquad
\begin{array}{r}
1\ 1\ 1\ 1\ \ \\
1\ 1\ 0\ 1 \\
1\ 0\ 1\ 1 \\
\hline
1\ 1\ 0\ 0\ 0
\end{array}$$

no overflow

Circuits for binary addition

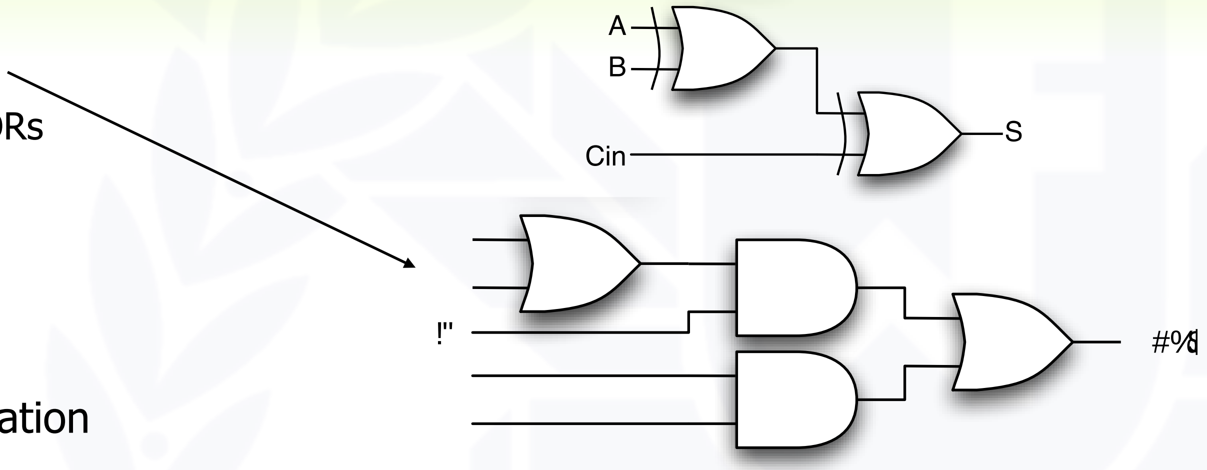
- Half adder (add 2 1-bit numbers)
  - Sum = Ai' Bi + Ai Bi' = Ai xor Bi
  - Cout = Ai Bi
- Full adder (carry-in to cascade for multi-bit adders)
  - Sum = Ci xor A xor B
  - Cout = B Ci + A Ci + A B = Ci (A + B) + A B

Ai	Bi	Sum	Cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

Ai	Bi	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

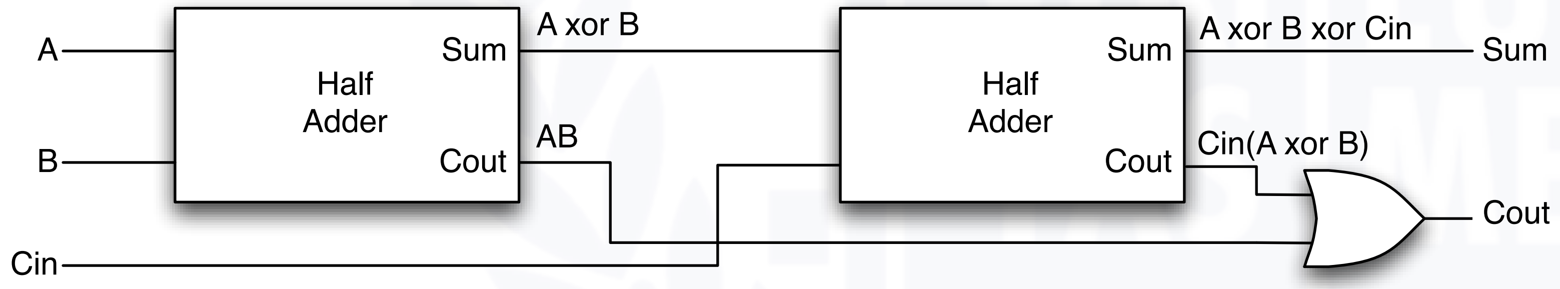
Full adder implementations

- Standard approach
  - 6 gates
  - 2 XORs, 2 ANDs, 2 ORs



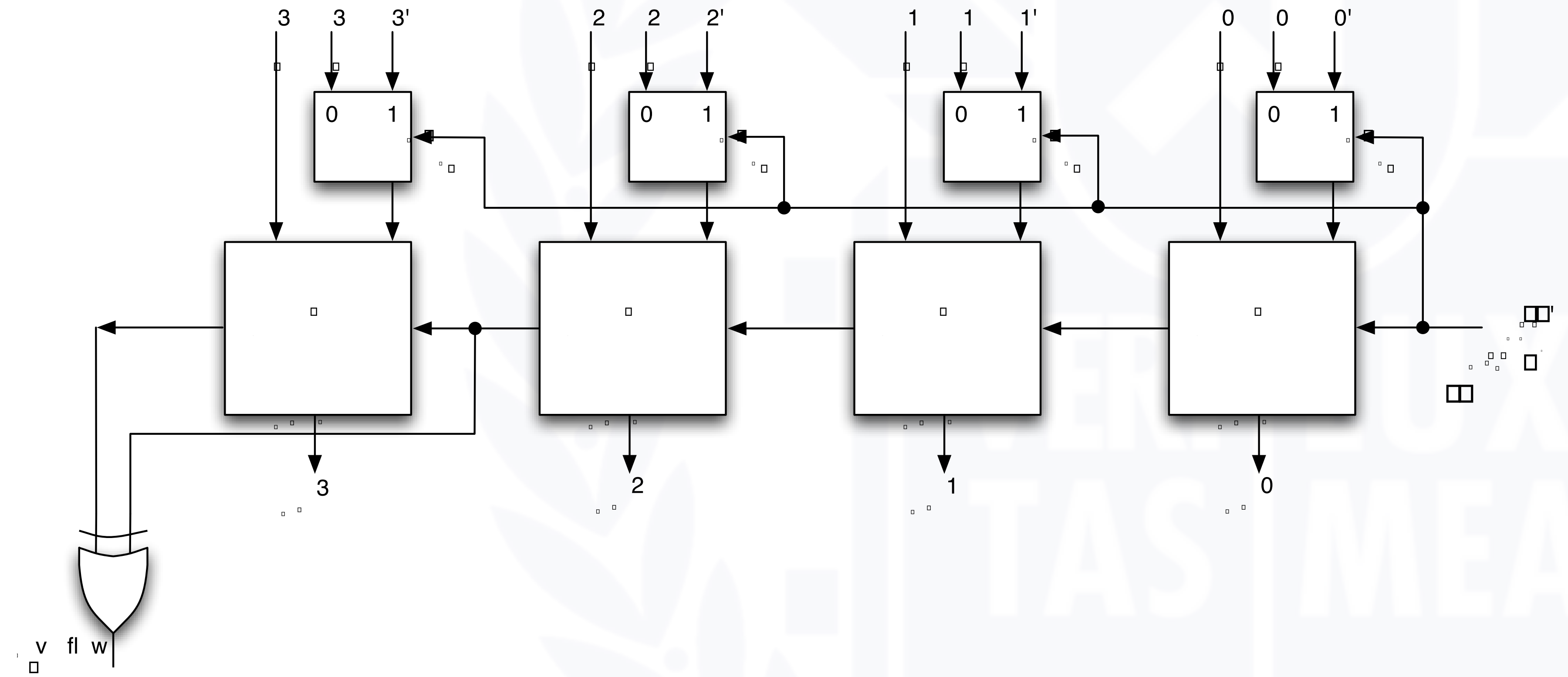
- Alternative implementation
  - 5 gates
  - Half adder is an XOR gate and AND gate
  - 2 XORs, 2 ANDs, 1 OR

Cout = A B + Cin (A xor B) = A B + B Cin + A Cin



Adder/subtractor

- Use an adder to do subtraction thanks to 2s complement representation
  - A – B = A + (– B) = A + B' + 1
  - Control signal selects B or 2s complement of B

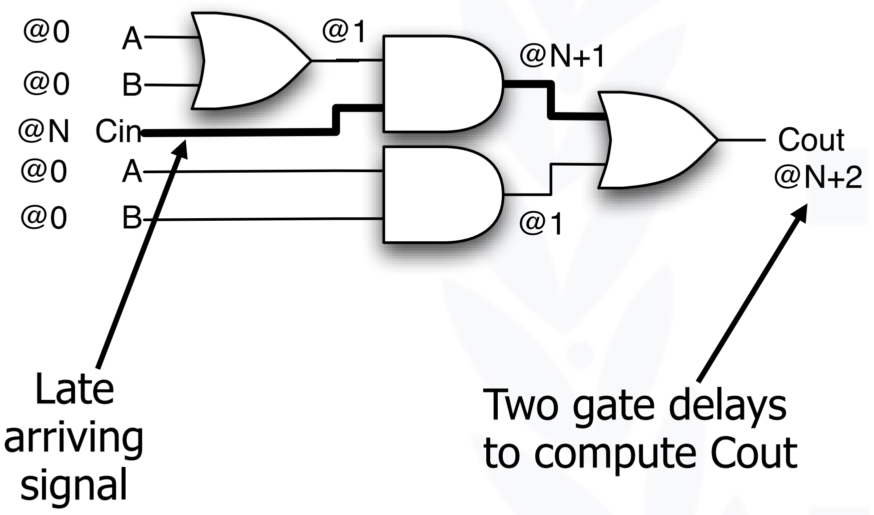


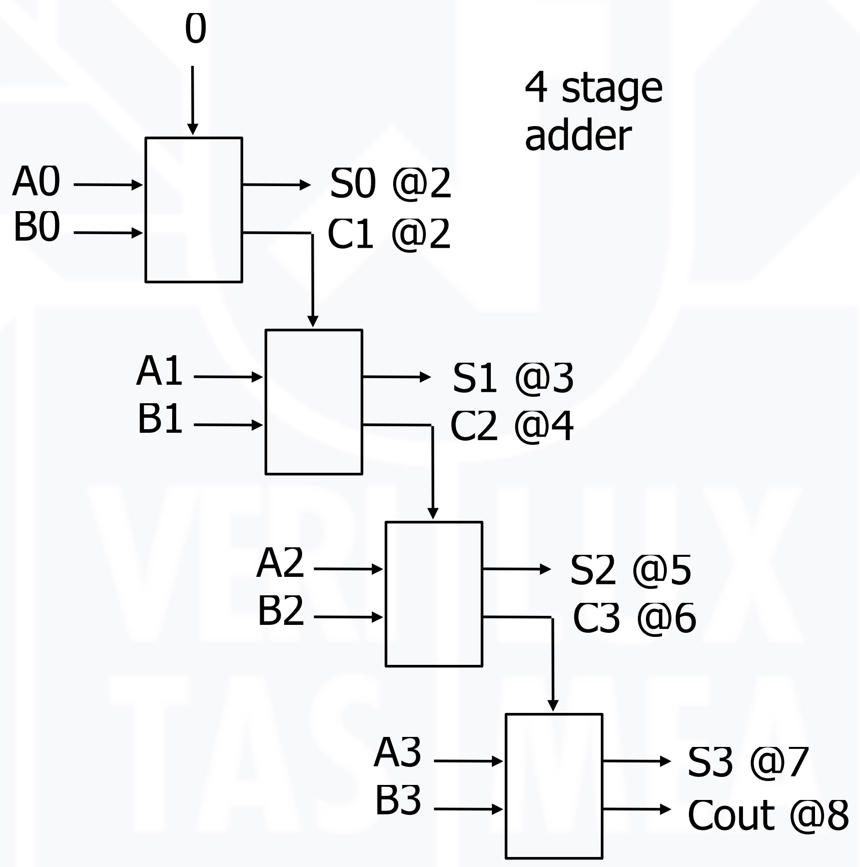
Embedded Low-Power Laboratory

Monday, March 15, 2010

Ripple-carry adders

- Critical delay
  - The propagation of carry from low to high order stages

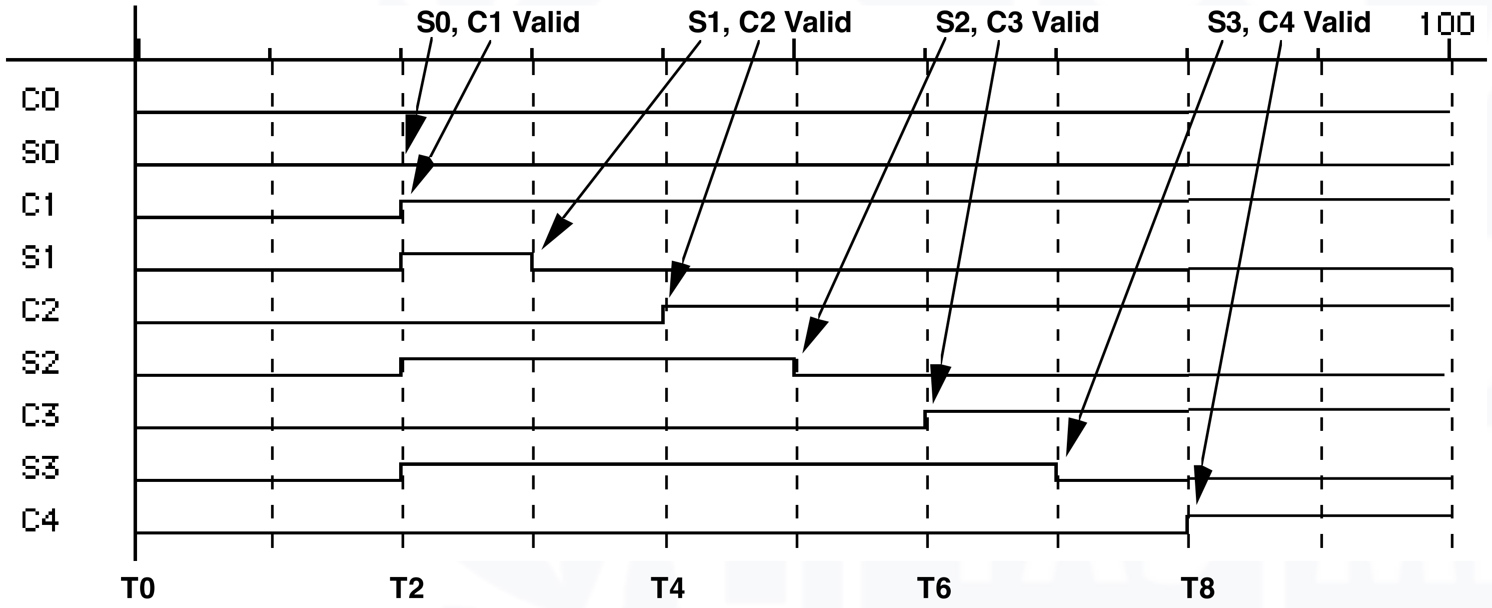




Embedded Low-Power Laboratory

Ripple-carry adders (cont'd)

- Critical delay
  - The propagation of carry from low to high order stages
  - 1111 + 0001 is the worst case addition
  - Carry must propagate through all bits



Carry-lookahead logic

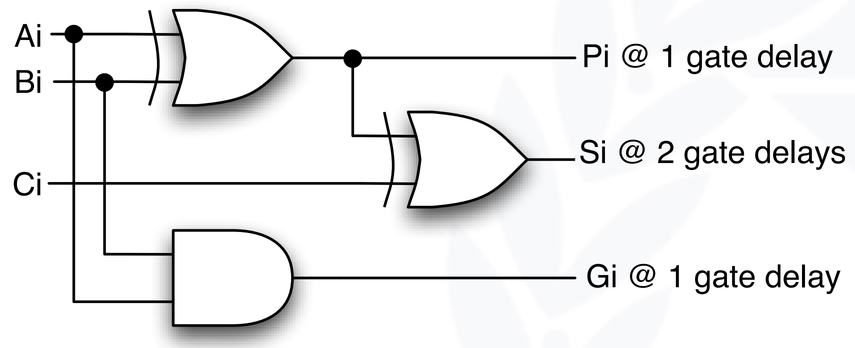
- Carry generate: Gi = Ai Bi
  - Must generate carry when A = B = 1
- Carry propagate: Pi = Ai xor Bi
  - Carry-in will equal carry-out here
- Sum and Cout can be re-expressed in terms of generate/propagate:
  - Si = Ai xor Bi xor Ci
    = Pi xor Ci
  - Ci+1 = Ai Bi + Ai Ci + Bi Ci
    = Ai Bi + Ci (Ai + Bi)
    = Ai Bi + Ci (Ai xor Bi)
    = Gi + Ci Pi

Carry-lookahead logic (cont'd)

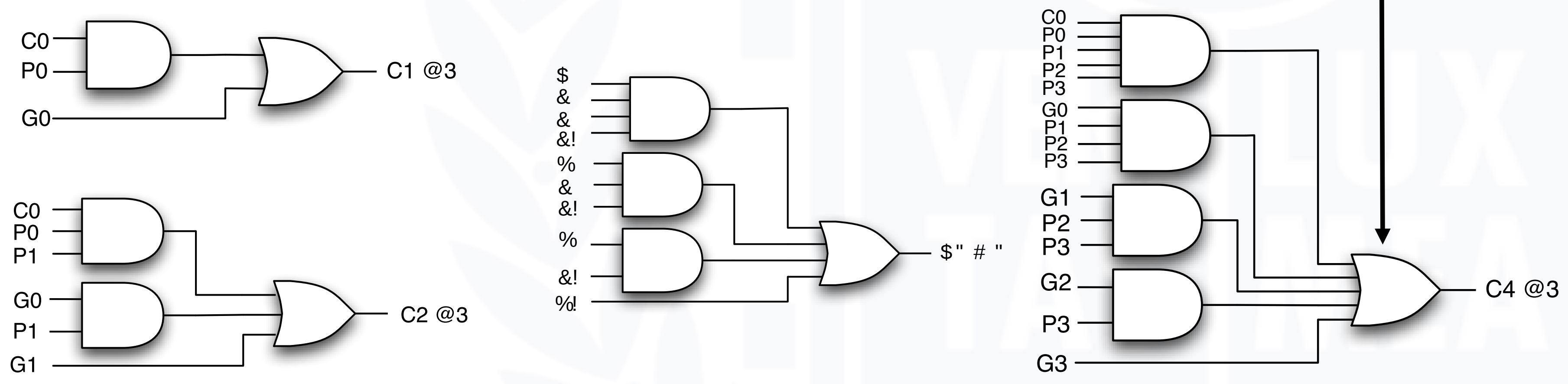
- Re-express the carry logic as follows:
  - C1 = G0 + P0 C0
  - C2 = G1 + P1 C1 = G1 + P1 G0 + P1 P0 C0
  - C3 = G2 + P2 C2 = G2 + P2 G1 + P2 P1 G0 + P2 P1 P0 C0
  - C4 = G3 + P3 C3 = G3 + P3 G2 + P3 P2 G1 + P3 P2 P1 G0 + P3 P2 P1 P0 C0
- Each of the carry equations can be implemented with two-level logic
  - All inputs are now directly derived from data inputs and not from intermediate carries
  - This allows computation of all sum outputs to proceed in parallel

Carry-lookahead implementation

- Adder with propagate and generate outputs

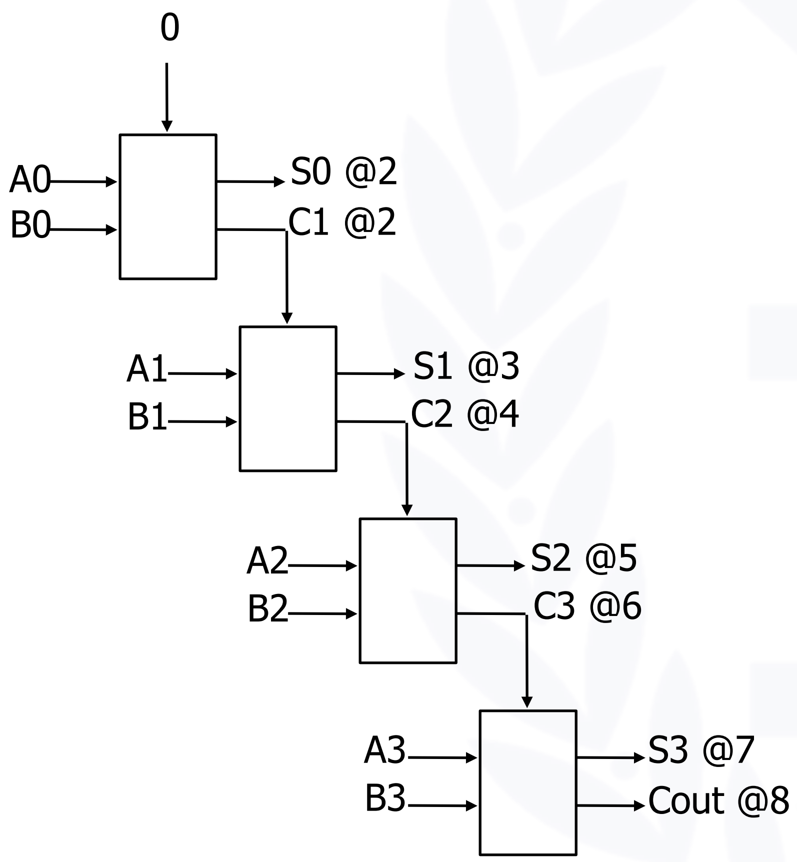


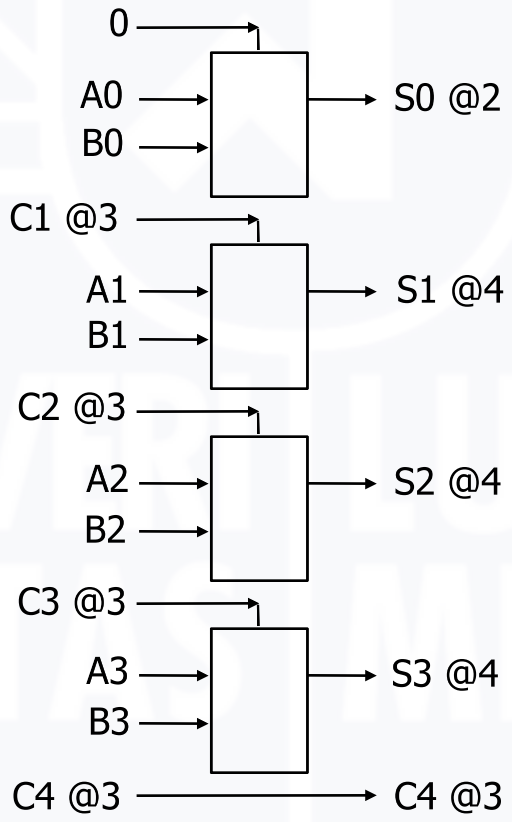
Increasingly complex logic for carries



Carry-lookahead implementation (cont'd)

- Carry-lookahead logic generates individual carries
  - Sums computed much more quickly in parallel
  - However, cost of carry logic increases with more stages

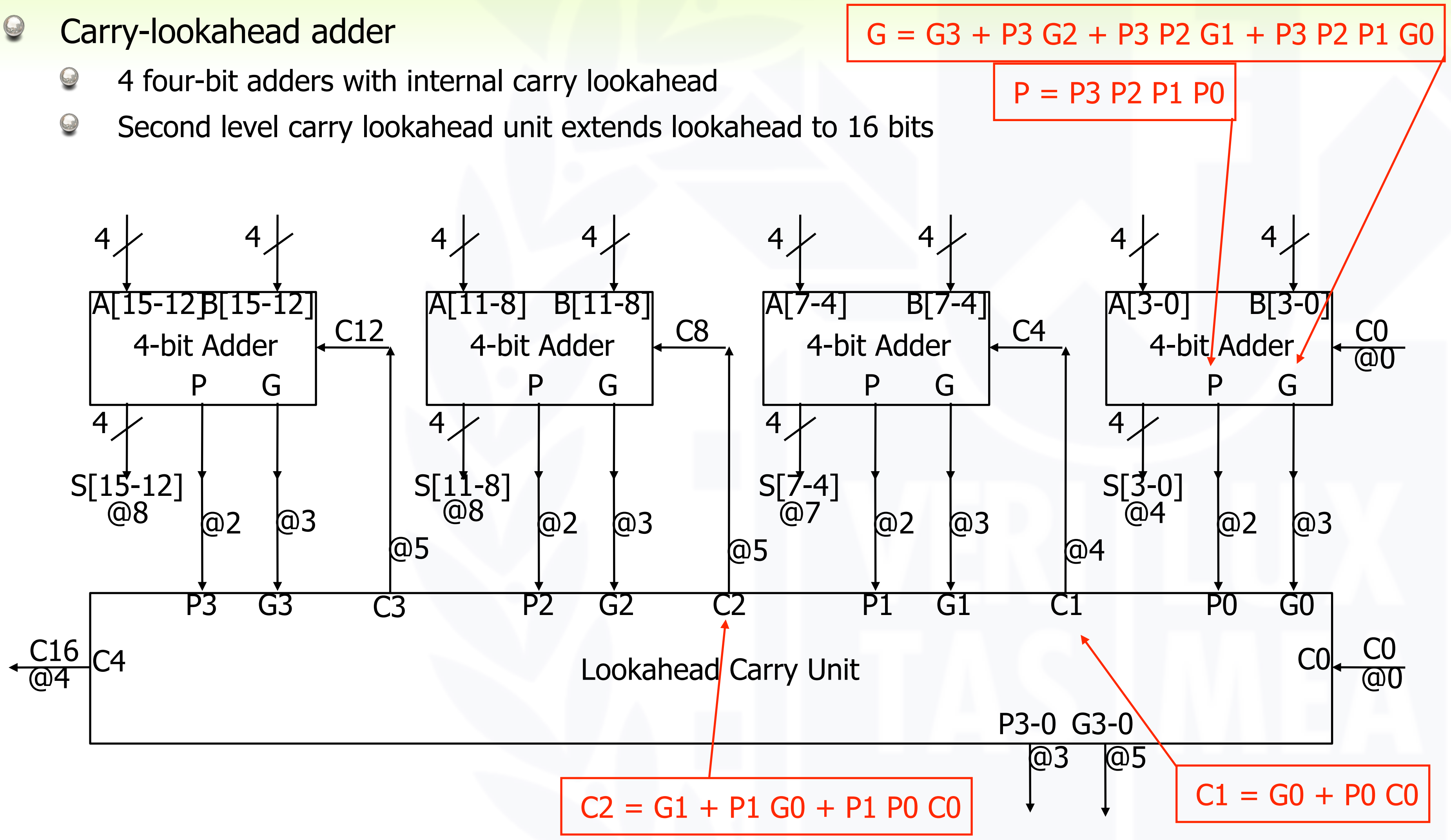




Embedded Low-Power Laboratory

Carry-lookahead adder with cascaded carry-lookahead logic

- Carry-lookahead adder
  - 4 four-bit adders with internal carry lookahead
  - Second level carry lookahead unit extends lookahead to 16 bits

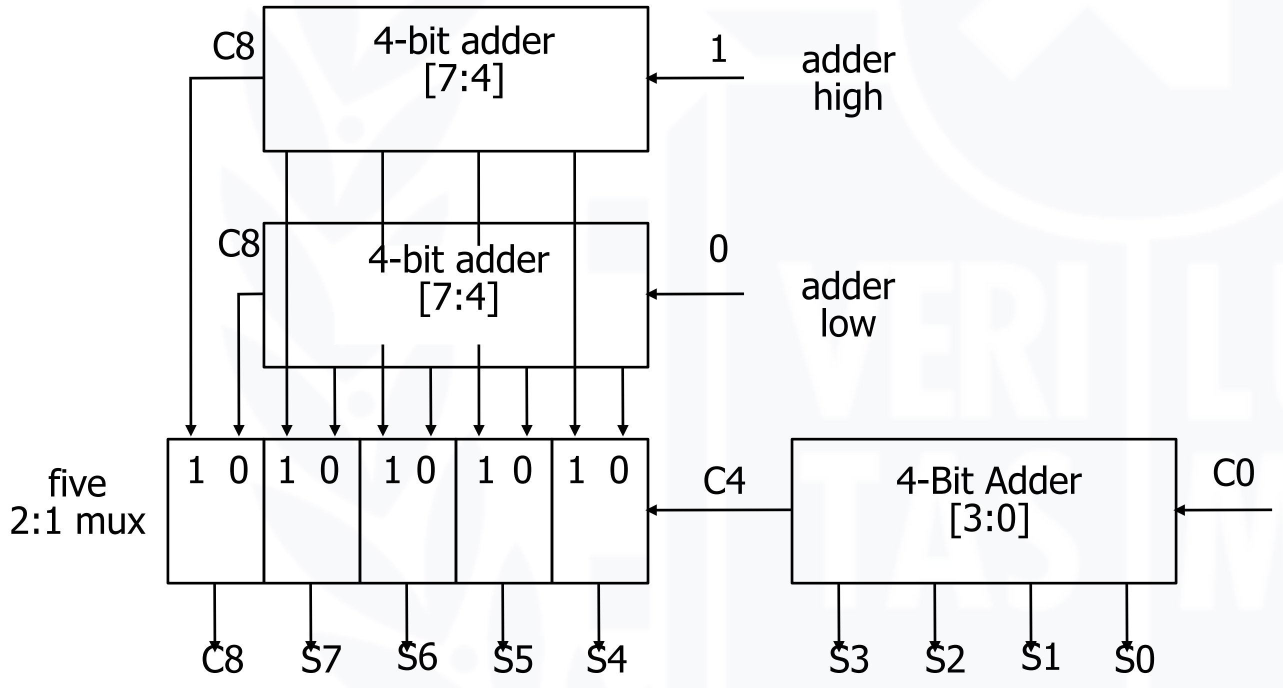


Embedded Low-Power Laboratory

Monday, March 15, 2010

Carry-select adder

- Redundant hardware to make carry calculation go faster
  - Compute two high-order sums in parallel while waiting for carry-in
  - One assuming carry-in is 0 and another assuming carry-in is 1
  - Select correct result once carry-in is finally computed



Arithmetic logic unit design specification

M = 0, logical bitwise operations

S1	S0	Function	Comment
0	0	Fi = Ai	input Ai transferred to output
0	1	Fi = not Ai	complement of Ai transferred to output
1	0	Fi = Ai xor Bi	compute XOR of Ai, Bi
1	1	Fi = Ai xnor Bi	compute XNOR of Ai, Bi

M = 1, C0 = 0, arithmetic operations

0	0	F = A	input A passed to output
0	1	F = not A	complement of A passed to output
1	0	F = A plus B	sum of A and B
1	1	F = (not A) plus B	sum of B and complement of A

M = 1, C0 = 1, arithmetic operations

0	0	F = A plus 1	increment A
0	1	F = (not A) plus 1	twos complement of A
1	0	F = A plus B plus 1	increment sum of A and B
1	1	F = (not A) plus B plus 1	B minus A

Logical and arithmetic operations
not all operations appear useful, but "fall out" of internal logic

Arithmetic logic unit design (cont'd)

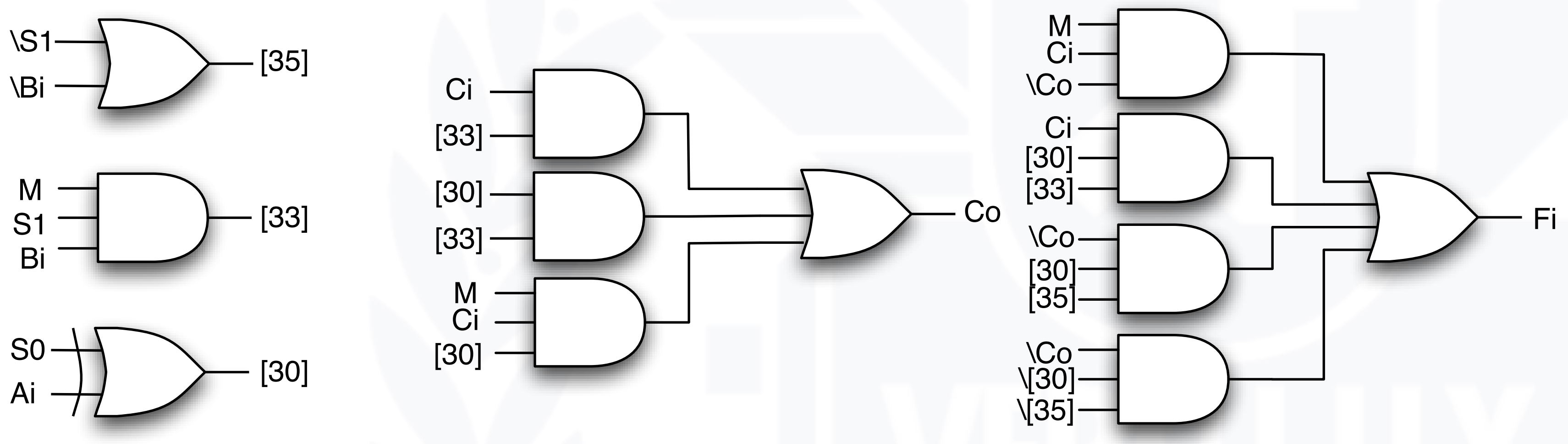
- Sample ALU – truth table

M	S1	S0	Ci	Ai	Bi	Fi	Ci+1
0	0	0	X	0	X	0	X
			X	1	X	1	X
	0	1	X	0	X	1	X
			X	1	X	0	X
	1	0	X	0	0	0	X
			X	0	1	1	X
			X	1	0	1	X
			X	1	1	0	X
	1	1	X	0	0	1	X
			X	0	1	0	X
			X	1	0	0	X
			X	1	1	1	X
1	0	0	0	0	X	0	X
			0	1	X	1	X
	0	1	0	0	X	1	X
			0	1	X	0	X
	1	0	0	0	0	0	0
			0	0	1	1	0
			0	1	0	1	0
			0	1	1	0	1
	1	1	0	0	0	1	0
			0	0	1	0	1
			0	1	0	0	0
			0	1	1	1	0
1	0	0	1	0	X	1	0
			1	1	X	0	1
	0	1	1	0	X	0	1
			1	1	X	1	0
	1	0	1	0	0	1	0
			1	0	1	0	1
			1	1	0	0	1
			1	1	1	1	1
	1	1	1	0	0	0	1
			1	0	1	1	1
			1	1	0	1	0
			1	1	1	0	1

Embedded Low-Power Laboratory

Arithmetic logic unit design (cont'd)

- Sample ALU – multi-level discrete gate logic implementation

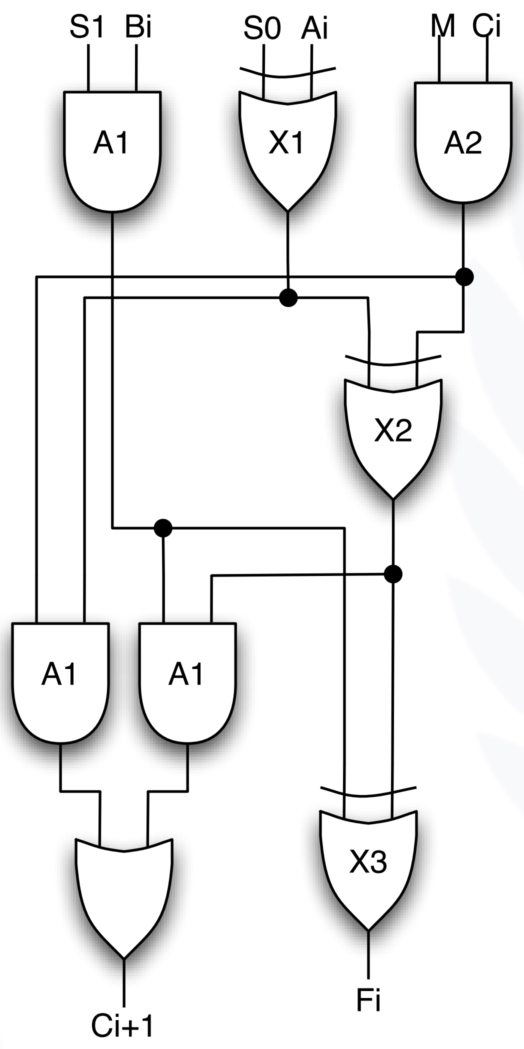


12 gates

Embedded Low-Power Laboratory

Arithmetic logic unit design (cont'd)

- Sample ALU – clever multi-level implementation



First-level gates
  use S0 to complement Ai
    S0 = 0 causes gate X1 to pass Ai
    S0 = 1 causes gate X1 to pass Ai'
  use S1 to block Bi
    S1 = 0 causes gate A1 to make Bi go forward as 0 (don't want Bi for operations with just A)
    S1 = 1 causes gate A1 to pass Bi
  use M to block Ci
    M = 0 causes gate A2 to make Ci go forward as 0 (don't want Ci for logical operations)
    M = 1 causes gate A2 to pass Ci

Other gates
  for M=0 (logical operations, Ci is ignored)
    Fi = S1 Bi xor (S0 xor Ai)
      = S1'S0' (Ai) + S1'S0 (Ai') +
        S1 S0' (Ai Bi' + Ai' Bi) + S1 S0 (Ai' Bi' + Ai Bi)
  for M=1 (arithmetic operations)
    Fi = S1 Bi xor ((S0 xor Ai) xor Ci) =
    Ci+1 = Ci (S0 xor Ai) + S1 Bi ((S0 xor Ai) xor Ci) =

    just a full adder with inputs S0 xor Ai, S1 Bi, and Ci

Summary for examples of combinational logic

- Combinational logic design process
  - Formalize problem: encodings, truth-table, equations
  - Choose implementation technology (ROM, PAL, PLA, discrete gates)
  - Implement by following the design procedure for that technology
- Binary number representation
  - Positive numbers the same
  - Difference is in how negative numbers are represented
  - 2s complement easiest to handle: one representation for zero, slightly complicated complementation, simple addition
- Circuits for binary addition
  - Basic half-adder and full-adder
  - Carry lookahead logic
  - Carry-select
- ALU Design
  - Specification, implementation